Building a DBMS on top of the JuxMem Grid Data-Sharing Service

Abdullah Almousa Almaksour[†], Gabriel Antoniu[†], Luc Bougé[‡], Loïc Cudennec[†], Stéphane Garçanski[§]
[†]INRIA/IRISA, Campus de Beaulieu, 35042 Rennes cedex, France
[‡]ENS Cachan/Antenne de Bretagne, Campus Ker Lann, 35170 Bruz, France
[§]LIP6 - Université Pierre et Marie Curie 6, 104 avenue du Président Kennedy, 75016 Paris France

Contact: gabriel.antoniu@irisa.fr

***Abstract*— We claim that building a distributed DBMS on top of a general-purpose grid data-sharing service is a natural extension of previous approaches based on the distributed shared memory paradigm. The approach we propose consists in providing the DBMS with a transparent, persistent and fault-tolerant access to the stored data, within a unstable, volatile and dynamic environment. The DBMS is thus alleviated from any concern regarding the dynamic behavior of the underlying nodes. We report on a feasibility study carried out with our JuxMem grid data-sharing service built on top of the JXTA peer-to-peer platform.**

***Index Terms*— Database, distributed system, grid, fault tolerance, JuxMem, JXTA.**

I. Introduction

Database Management Systems (DBMS) are designed to enable application programs to store and retrieve structured data, while being protected from details of data representation and storage. In particular, the applications should not bother about data integrity and security, concurrent access and crash recovery, data administration, physical structure of the network of storing nodes, etc.

Most of the existing DBMS are nowadays classified as *on-disk resident DBMS*. That means that the files permanently storing the database reside on hard disks. Of course, every such DBMS has a specific technique for *intelligently* caching some parts of these data files within the main RAM memory, in order to accelerate the data access. However, any data modification must be validated on the hard disk copy in a way or another before completing. The underlying physical node is a critical point in the system. When it is down, the whole system will be blocked. Furthermore, regarding this centralized aspect, the processing of individual transactions is essentially sequential.

To improve fault tolerance, replication techniques have been used in the context of *distributed databases* [1]. These techniques have also opened the way for performance improvements through parallel query processing techniques. On the other hand, to lower the data access cost, main memory DBMS have been proposed, relying on the idea of storing the whole database in the main memory. Finally, both approaches have been combined in the design of *distributed main-memory databases*.

In a totally different context, data sharing using distributed, in-memory storage has been used for parallel and distributed scientific computing. More specifically, *distributed shared memory systems* (DSM) [2] have been used to transparently share data at the scale of a cluster. These systems favored programming simplicity: the application only has to read/write the data using global identifiers, without having to manage explicitly data localization and transfer among the distributed nodes. Initially developed at small scales (i.e. on clusters of a few tens of nodes), this approach has been extended to a grid scale (thousands of nodes) by the concept of *grid data-sharing service*, a hybrid system which combines the benefits of distributed shared memory systems (transparent access, data consistency) with those of peer-to-peer systems (volatility tolerance, scalability) [3]. This concept is illustrated by the JuxMem platform [4].

In this paper we explore the idea of building a scalable, grid-enabled database management system, by combining the concepts of *main-memory databases* and *grid data-sharing service*. The goal is to provide facilities for managing structured data through a relational database language on top of a large-scale, dynamic grid infrastructure. This can be achieved by storing the database using the grid data-sharing service, which transparently manages data distribution, localization, replication and transfer, in a fault-tolerant way.

The remaining of this paper is structured as follows. The next section discusses previous related work. Section III describes our approach. Implementation details and a very preliminary evaluation are given in Section IV. Finally, Section V summarizes the contribution and gives directions for future work.

II. Related work

In previous work in the field of databases, various techniques have been proposed in order to improve the cost of the database operations and the tolerance to failures of critical points.

a) Distributed Databases: To improve performance, it has of course been proposed to implement the database on top of a cluster of nodes, to use *parallel* request evaluation techniques and physical data *distribution* [1]. At the same time, it is possible to use the multiple nodes to replicate the functionalities (i.e. the data) and thereby remove the single point of failure.

b) Main Memory Databases: Thanks to the increasing main memory capacity, several research efforts aimed to take benefit from this development to further lower the cost of database accesses. The most popular idea was to fit the whole database in the main memory, in such a way that the hard disk access during data manipulation could be partially eliminated. Such systems have then been called "memory resident databases" or "Main Memory Databases" (MMDB)[5]. The most difficult challenge in such systems was to find a way to store permanent data in non permanent, volatile storage device (main memory). To achieve this, some MMDB systems required specific hardware designs, others adapted some logging techniques, where all data changes must be logged - sooner or later- on the hard disk. Several optimization algorithms were developed for this purpose [6], [7].

c) Distributed, Main Memory Databases: Finally, some systems combined the two approaches mentioned above and addressed the challenge of building a distributed, memory-based database management system trying to benefit both from fast RAM accesses and from parallel processing of distributed data. PRISMA/DB [8] is such a relational DBMS that achieves high performance through parallel query processing and by using main memory storage of the entire database. PRISMA/DB is implemented on a parallel multiprocessor (the POOMA machine). Similarly, DERBY [9] makes use of existing general purpose workstations interconnected via a high-speed network to create a massive, cost-effective main memory storage system. The network will be restricted to a local area radius to limit the network latency between any two machines. Finally, Sprint [10] is an in-memory database middleware implemented on cluster of servers. Sprint partitions and replicates the database into segments and stores them in several data servers. Sprint supports distributed query processing. Each data server is responsible for running a local in-memory database, so it receives queries concerning its own database and executes them without any disk access. Some special nodes (durability servers) ensure on-disk transaction logging and manage the recovery process in case of failure of the data servers. Sprint has been tested on a cluster consisting of 64 computing nodes.

d) Distributed Shared Memory (DSM) systems: In parallel to the efforts mentioned above, which specifically target database applications, other research efforts have explored the use of distributed main-memory storage in a more generic way. Distributed Shared Memory systems in particular, such as Ivy [11] or TreadMarks [12] provided the illusion of a unique, globally shared memory on top of a set of physically distributed memories. Used in the field of high-performance parallel computing, such systems provide the various computing processes that are part of the application with the ability to uniformly access any data, independently of the data location. The DSM system automatically handles data localization, transfer or replication. In this context, multiple models have been proposed (sequential consistency, release consistency, etc.), in order to enable different trade-offs between the strictness of the consistency guarantees and the performance of the underlying protocols that ensure the consistency of the various data copies. As opposed to databases, which provide a way to manage and query structured data, DSM systems allow direct access to unstructured data through direct reads or writes.

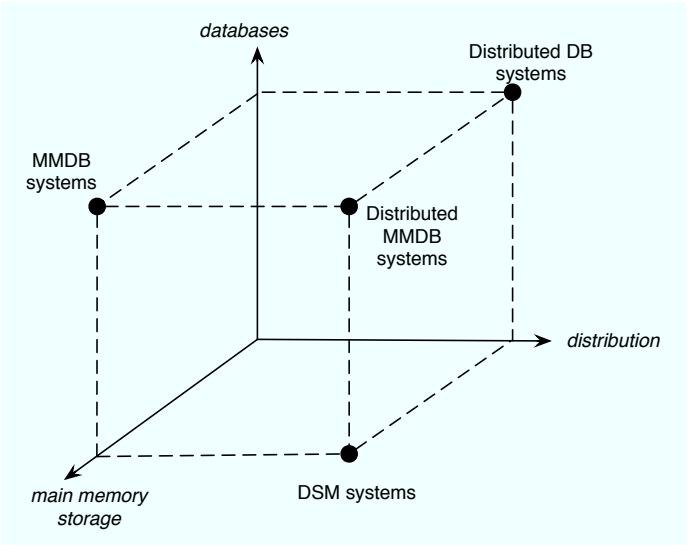

Fig. 1. Various classes of data management systems.

Figure 1 provides a global picture illustrating the positioning of the various systems discussed above with respect to three development axes: from centralized to distributed architectures, from disk-based storage to main memory storage; from unstructured data access to structured data management.

III. Our proposal: building a DBMS on top of a grid data-sharing service

As a continuation of previous efforts aiming at providing DBMS facilities over a cluster of workstations, in this work we explore the possibility of building such a system at a larger scale, on top of a grid infrastructure.

A. Towards transparent data access: the JuxMem grid data-sharing service

Currently, the most widely-used approach to manage data on distributed environments (and on grid testbeds in particular) relies on the *explicit data access model*, where clients have to move data to computing servers. In order to achieve a real virtualization of the management of large-scale distributed data, a step forward has been

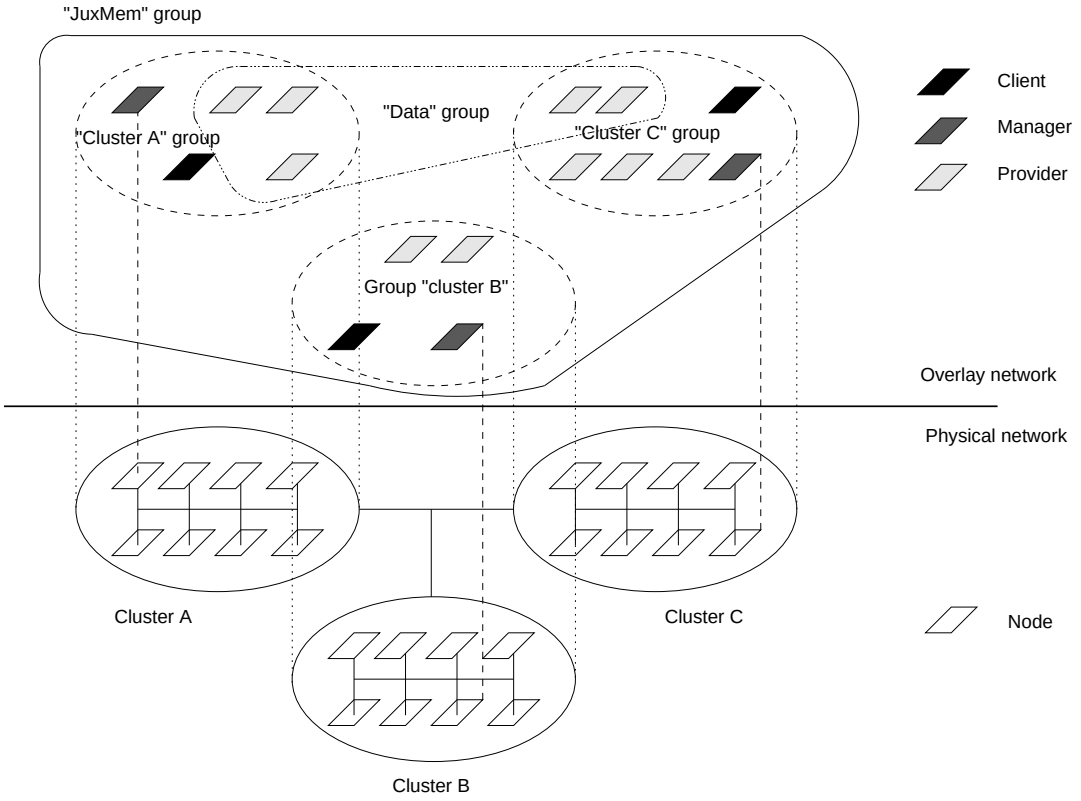

Fig. 2. Hierarchy of the entities in the network overlay defined by JuxMem.

made by enabling a *transparent data access model* through the concept of *grid data-sharing service* [4]. Such a service transparently manages data localization and persistence in a dynamic, large-scale, distributed environment. The data-sharing service concept is based on a hybrid approach inspired by DSM systems and peer-to-peer (P2P) systems. The service provides data consistency guarantees based on fault-tolerant protocols that keep data replicas consistent despite possible failures and disconnections [13]

The concept of data-sharing service is illustrated by the JuxMem software experimental platform, described in detail in [4]. Its implementation relies on the JXTA [14] generic P2P framework. JuxMem's architecture mirrors a grid consisting of a federation of distributed clusters and it is therefore expressed in terms of *hierarchical* groups. The goal is to accurately take into account the latency hierarchy of the physical network topology, to take advantage of the low-latency links within the clusters and reduce higher-latency, inter-cluster communications. All nodes participating to the data-sharing service network overlay are members of the juxmem group. All members of the juxmem group that belong to the same physical cluster form a cluster group.

Any cluster group consists of *provider* nodes which supply memory for data storage. Any node (including providers) may use the service to allocate, read or write data as *clients*, in a peer-to-peer approach.

The JuxMem API provides users with classical functions for memory allocation in the globally shared space: `juxmem_malloc`, `juxmem_calloc`, etc. When allocating memory, a client has to specify on how many sites[1] the data should be replicated, and on how many nodes in each site. This results into the instantiation of a set of data replicas, associated to a group of peers called *data group*. The memory allocation operation returns a global data ID which can be used by other nodes to get access to the corresponding data, through the use of the juxmem_mmap primitive. Before reading or writing the data, a process that uses JuxMem should acquire the lock associated to the data through either `juxmem_acquire` (for read and write access) or `juxmem_acquire_read` (for read-only access). This API allows the implementation to apply consistency guarantees according to the consistency protocol specified by the user at allocation time.

The data group is also hierarchically organized: it consists of a *Global Data Group (GDG)* which gathers all provider nodes holding a replica of the same piece of data. These nodes can be distributed in different sites (clusters), thereby increasing the data availability if faults occur. The GDG is split into *Local Data Groups (LDG)*, which correspond to data copies located in a same site. The GDG acts as a self-organizing group able to adapt itself in case of dynamic changes due to possible failures (by creating new replicas or removing old ones).

[1]A site is a set of nodes, e.g.. a physical cluster within a cluster federation.

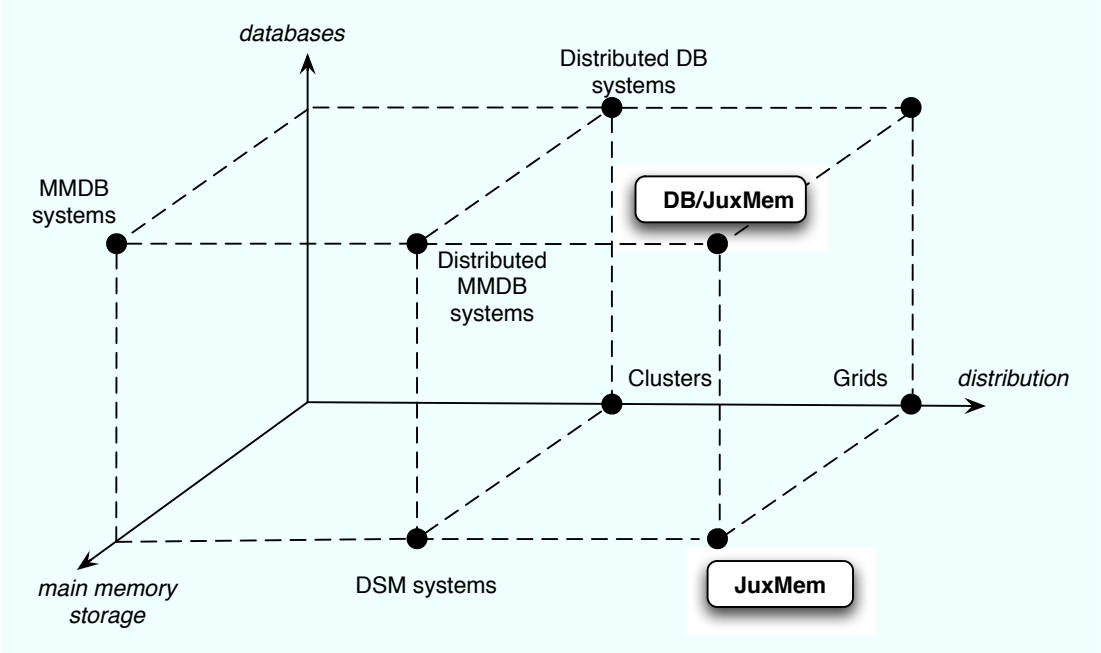

Fig. 3. Positioning of the proposed JuxMem-based grid-enabled DBMS

B. A DBMS over JuxMem: architecture overview

All previous efforts which aimed at providing a distributed memory DBMS reached at best the scale of a cluster of computers of up to a few tens of nodes.If we assume an average RAM size of 2 GB at every node, the overall memory space typically gathers together 120 GB. This may be suitable for several applications, but may not be enough for databases that need to host large masses of information (e.g. produced by large-scale physical simulations).

To get a better scalability, we rather propose to target a grid computing infrastructure (gathering up to several thousands of nodes), by leveraging the benefits of a grid data-sharing service such as JuxMem. We can see our contribution from two points of view. First, our approach allows to build a high-performance, scalable DBMS over a dynamic, distributed infrastructure by delegating to the grid data-sharing service all aspects related to data localization, transfer, replication, etc., which are handled in a fault tolerant way. Second, another goal is to extend JuxMem with a DB-oriented API enabling scientific application running over JuxMem to manage their data in structured manner. *Figure 3* illustrates the position of our proposed approach.

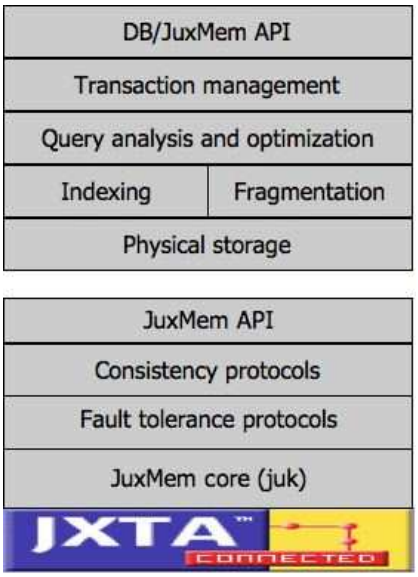

Fig. 4. DB/JuxMem Layered Architecture.

As JuxMem currently manipulates data only as a sequence of bytes, more sophisticated high-level layers over JuxMem become necessary in order to propose an API allowing to create, modify, query a database. The *Physical Storage* layer uses JuxMem's API to store data blocks, which can represent the various parts of the database within JuxMem: data, indexes, table schemas, database catalogue, etc. The *Index* layer manages table indexing. It uses the physical storage primitives for creating and updating the indexes. The *Fragmentation* layer allows to split large tables into multiple fragments, for an efficient storage and management with JuxMem. Then, the *Query analysis and optimization* layer and the *Transaction management* layer handle higher-level aspects of database operations. Finally, a DB-oriented API for JuxMem gives access to structured data management on top of JuxMem.

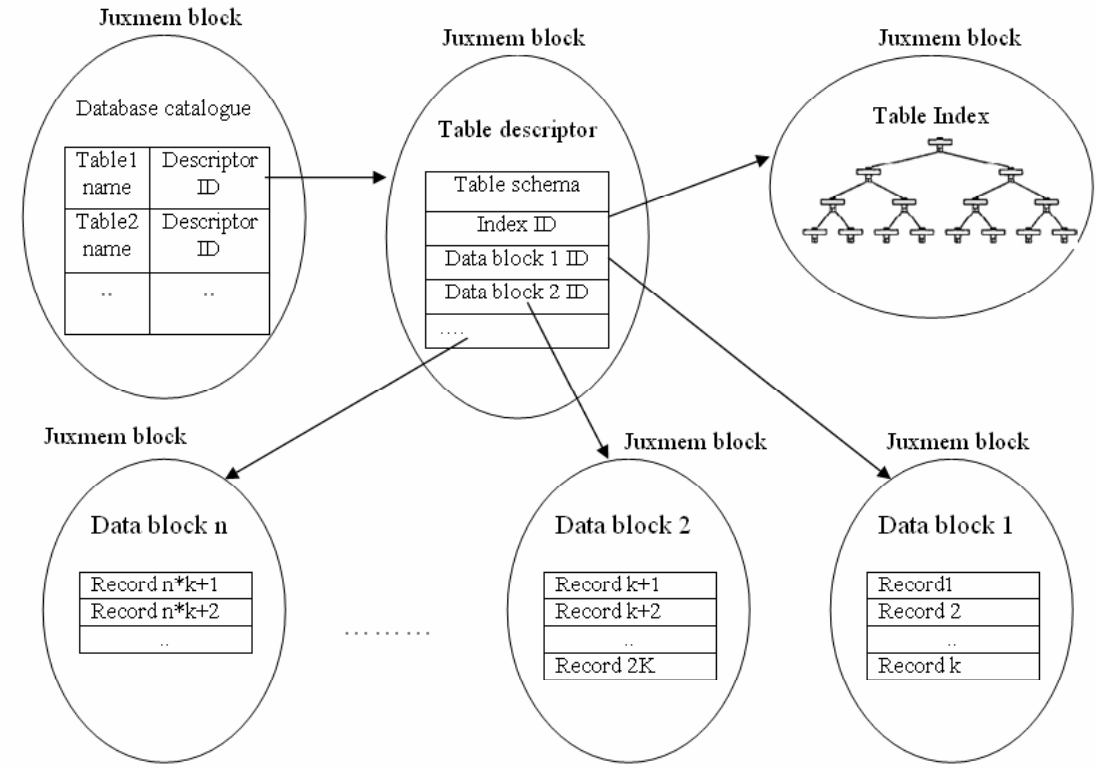

Fig. 5. Database representation in DB/JuxMem.

IV. Feasibility study: implementation and evaluation

The implementation of the *Query analysis* layer and of the *Transaction* layer are beyond the scope of our intended contribution. At this early stage, we focus on providing support for basic queries (no transactions are supported yet). This requires the physical storage, indexing and fragmentation layers to be operational. The implementation has been realized using the C programming language, based on JuxMem 0.4.

For each database created by DB-JuxMem, a special data catalogue is stored as a JuxMem block. This catalogue consists of an array of records, each of which has two fields: a table name and the ID of a JuxMem block containing the descriptor of this table (see below). When creating a new table, at least three new JuxMem blocks are allocated: one for the table descriptor, one for the table index and one (at least) for the table records.

The table descriptor contains the following information (Figure 5):

- The **table schema**, as specified by the user;
- The **data block IDs**; The table records will be distributed over several memories over the grid using JuxMem. Each set of k consecutive records are stored together in a JuxMem block, using horizontal fragmentation. When creating a new data block, its JuxMem ID is stored in the table descriptor.
- The **index ID**. An index is built for each table. The index value is the primary key defined by the user. In this index, we associate each key value with 1)the ID of the data block which contains the record; 2)the *offset* of this record in the data block. The table index is stored in a separate JuxMem block, and the address of this block is stored in the table descriptor. For the sake of simplicity, in this first implementation we used a balanced binary tree, however we plan to explore in the future the possible benefits of using a T-tree structure, which has been shown to meet the needs of main memory databases [15].

Based on the data structures described above and relying on JuxMem's memory allocation and data access primitives, we have implemented a runtime-level, DB-oriented API for database creation, table creation, table update, etc. This API could be used as a glue in order to allow existing DBMS to benefit from JuxMem's properties, such as a transparent, fault-tolerant storage, distributed at a grid scale. A selection of these primitives is presented in Table I.

To validate our study, we performed a preliminary experimental evaluation on a cluster of 2.3 GHz Intel Xeon 5148 LV nodes with 4 GB RAM, interconnected by a regular Gigabit Ethernet local area network. This cluster is part of Grid5000, a reconfigurable, controllable and monitorable experimental Grid platform gathering 9 sites geographically distributed in France. We measured the time it takes for DB/JuxMem to create a new table, to insert a variable number of records and to search a specific value in a table. We provide here the results for table creation, where the table is stored on a cluster

Primitive name	Description
DBJuxMemDBCreate	Create the database
DBJuxMemTableCreate	Create a table and store it into JuxMem.
DBJuxMemTableMap	Map an existing table to local memory
DBJuxMemTableUnMap	Unmap an existing table from local memory
DBJuxMemTableAcquire	Get exclusive access to a table
DBJuxMemTableReadAcquire	Get read-only access to a table
DBJuxMemTableRecordInsert	Insert a record into the table
DBJuxMemTableRecordDelete	Delete a record from the table
DBJuxMemTableSelect	Perform a select query on a table
DBJuxMemTableRelease	Release the access to a table
DBJuxMemTableDelete	Delete a table from the database

TABLE I
A SELECTION OF DB/JUXMEM PRIMITIVES.

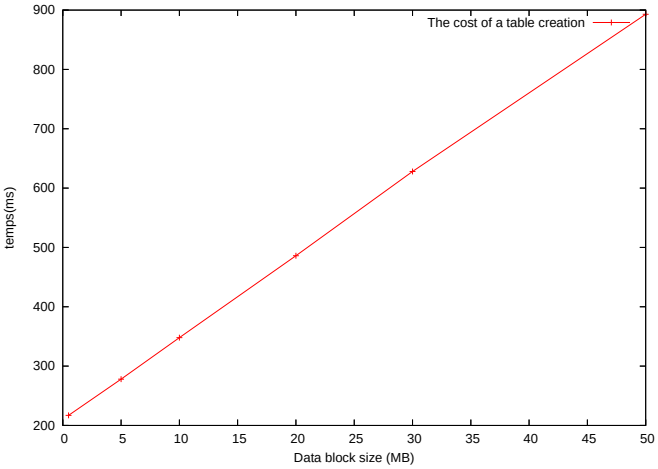

Fig. 6. Preliminary evaluation of the cost of a table creation

and replicated once on the same cluster (so, 2 replicas are allocated). Please note these figures only correspond to a preliminary feasibility study and do not reflect the best performance that can can be obtained through specific optimizations. As explained, the table creation requires to call `juxmem_malloc` several times, in order to allocate the table descriptor (fixed size), the table index (also set to a fixed maximum size) and the data block(s) (variable size). As expected, the cost is near linear with the data size.

V. CONCLUSION

This paper explores the idea of building a scalable, grid-enabled database management system by extending the concept of *main-memory database* with the facilities provided by a *grid data-sharing service*. This can be achieved by handling the database data and metadata using the grid data-sharing service, which transparently manages data allocation, distribution, localization, replication and transfer, in a fault-tolerant and consistent way.

We illustrate our approach by proposing a software architecture based on our JuxMem grid data-sharing service. The feasibility of the idea has been demonstrated by implementing a preliminary prototype allowing to perform basic database operations like table creation, record insertion and simple select queries.

Our current work focuses on the evaluation and optimization of the prototype, which will equally be extended to support more complex queries. As a second step, we will evaluate our approach in a larger-scale, multi-cluster grid configuration. Moreover, we plan to integrate our prototype (available as a runtime library) with an existing open-source main memory DBMS and to evaluate it with existing database applications. The main challenge is here to adequately choose storage and replication strategies for the DBMS components, in order to efficiently implement the required consistency semantics.

REFERENCES

[1] T. Oszu and P. Valduriez, *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.

[2] J. Protić, M. Tomasević, and V. Milutinović, *Distributed Shared Memory: Concepts and Systems*. IEEE, Aug. 1997.

[3] G. Antoniu, L. Bougé, and M. Jan, "Peer-to-peer distributed shared memory?" in *Proc. IEEE/ACM 12th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT 2003), Work in Progress Session*, New Orleans, Louisiana, September 2003, pp. 1–6. [Online]. Available: http://hal.inria.fr/inria-00000981/en

[4] G. Antoniu, L. Bougé, and M. Jan, "JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid," *Scalable Computing: Practice and Experience*, vol. 6, no. 3, pp. 45–55, Nov. 2005. [Online]. Available: http://www.irisa.fr/paris/Biblio/Papers/Antoniu/AntBouJan05SCPE.pdf

[5] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 509–516, 1992.

[6] T. J. Lehman and M. J. Carey, "A study of index structures for main memory database management systems," in *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 294–303.

[7] I. Lee and H. Y. Yeom, "A single phase distributed commit protocol for main memory database systems," in *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2002, p. 44.

[8] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut, "Prisma/db: A parallel, main memory relational dbms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 541–554, 1992.

[9] J. Griffioen, R. Vingralek, T. A. Anderson, and Y. Breitbart, "DERBY: A memory management system for distributed main memory databases," in *RIDE-NDS*, 1996, pp. 150–159. [Online]. Available: citeseer.ist.psu.edu/griffioen96derby.html

[10] M. W. Lasaro Camargos, Fernando Pedone, "Sprint: A middleware for high-performance transaction processing," in *Proceedings of the 2nd European Conference on Systems Research (EuroSys'2007)*, March 2007.

[11] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 321–359, Nov. 1989.

[12] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18–28, Feb. 1996.

[13] G. Antoniu, J.-F. Deverge, and S. Monnet, "How to bring together fault tolerance and data consistency to enable grid data sharing," *Concurrency and Computation: Practice and Experience*, no. 17, 2006, to appear. [Online]. Available: http://hal.inria.fr/inria-00000987

[14] "The JXTA Project," https://jxta.dev.java.net/.

[15] H. Lu, Y. Y. Ng, and Z. Tian, "T-tree or b-tree: Main memory database index structure revisited," in *ADC '00: Proceedings of the Australasian Database Conference*. Washington, DC, USA: IEEE Computer Society, 2000, p. 65.